УДК 004.724.4(045)

[1]**Давиденко И.Н.,** канд. техн. наук
[2]**Бабий К.Ю.**

ПРИМЕНЕНИЕ АСПЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА ПРИ РАЗРАБОТКЕ СОВРЕМЕННЫХ ПРОГРАММНЫХ СИСТЕМ

[1]**Институт компьютерных технологий**
Национального авиационного университета
[2]**ООО "ГлобалЛоджик Украина"**

Рассмотрены вопросы применения новой парадигмы программирования – аспектно-ориентированного подхода, основным свойством которой является модуляризация сквозных требований на разных уровнях абстракции и их локализация в отдельных программных модулях – аспектах

Введение

Аспектно-ориентированное программирование (АОП) представляет собой одну из концепций программирования, которая является дальнейшим развитием процедурного и объектно-ориентированного программирования (ООП). Данная методология призвана снизить время, стоимость и сложность разработки современного ПО. Как правило, можно выделить определенные части, или аспекты, отвечающие за ту или иную функциональность, реализация которой рассредоточена по коду программы, но состоит из схожих кусков кода. По оценкам специалистов [1], около 70% времени в проектах тратится на сопровождение и внесение изменений в готовый программный код. Поэтому достаточно важной в ближайшей перспективе становится роль АОП и подобных трансформационных подходов.

Обзор существующих решений

Сравнительно новая технология уже получила широкое распространение, показав свою эффективность в решения ряда задач.

Объектом исследования данной работы является аспектно-ориентированный подход при разработке программных систем. АОП предлагает языковые средства, позволяющие выделять сквозную функциональность в отдельные модули, и таким образом упрощать работу (отладку, модифицирование, документирование и т.д.) с компонентами программной системы, и снижать сложность системы в целом.

Основные идеи АОП были сформулированы в работе [2] идеологом методологии Г. Кикжалесом. Ими же в 2001 году было разработано первое и наиболее широко известное аспектно-ориентированное расширение языка программирования – *AspectJ* ().

В начале эры вычислительной техники программы разрабатывались посредством прямого кодирования на машинном языке. Программисты тратили значительно больше времени, раздумывая над особенностями использования того или иного набора машинных инструкций, чем непосредственно над стоящей перед ними задачей. Затем появились языки более высокого уровня, которые позволяли абстрагироваться от машинного уровня.

Потом пришла эра структурированных языков, когда программисты могли проводить структурную декомпозицию проблем в терминах процедур, необходимых для выполнения той или иной задачи. Тем не менее, с ростом сложности программного обеспечения возникла потребность в другой технологии.

Объектно-ориентированное программирование позволило представить систему как множество взаимодействующих объектов. Классы позволили скрыть детали реализации за интерфейсами. Механизм полиморфизма обеспечил общее поведение и интерфейс связанных концепций и позволил управлять поведением

компонентов без доступа к реализации базовых концепций.

В настоящее время объектно-ориентированное программирование (ООП) является методологией, в пользу которой делается выбор большинством новых проектов разработки программных продуктов. Несомненно, методология ООП продемонстрировала свою силу при моделировании общего поведения разрабатываемой системы. Однако, как можно убедиться из существующего опыта разработки программных продуктов, ООП не в достаточной мере позволяет справляться с растущей сложностью программных систем.

Постановка задачи

Как правило, программная система состоит из нескольких частей: основной (предметно-ориентированной) и системной части, которые несут в себе требуемую функциональность. Например, ядро системы обработки кредитных карт предназначено для работы с платежами, тогда как функциональность системного уровня предназначена для ведения журнала событий, целостности транзакций, авторизации, безопасности, производительности и т.д. Большинство подобных частей системы, известны как сквозная функциональность [3], затрагивают множество основных предметно-ориентиро-ванных модулей. Можно рассматривать сложную программную систему как комбинацию модулей, каждый из которых включает в себя кроме бизнес-логики часть сквозной функциональности из набора требований к системе. На рис. 1 представлена система как набор требований разбитых на разные модули.

При разработке программной системы с использованием существующих методологий программирования сквозная функциональность будет включена во все модули, в результате система будет сложной при проектировании, понимании, реализации и поддержке.

Современные технологии разработки ПО на уровне языков программирования предоставляют удобные средства для выделения логики функционирования программы в отдельные модули, но не одна из них не предлагает удобного способа локализации в отдельные модули функциональности, которая должна распространяться на всю систему.

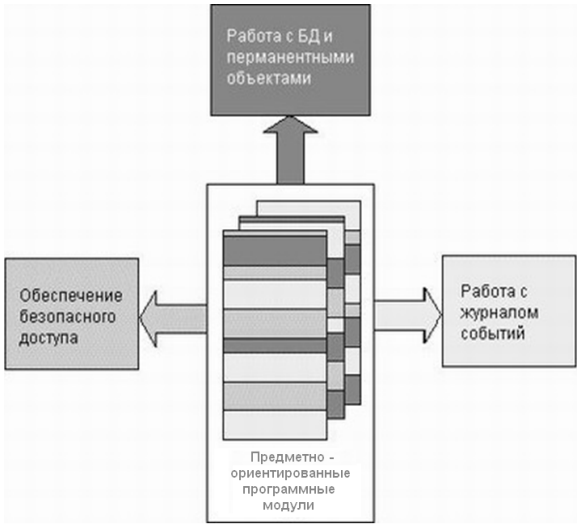

Рис. 1. Система как набор функциональных модулей

Из-за того, что реализация сквозной функциональности не может быть обособлена средствами языка программирования в отдельном программном модуле, элементы этой реализации присутствуют в том или ином виде в большинстве модулей, образующих программную систему.

Рассмотрим пример java-класса, основной задачей которого является реализация некоторой логики:

```
public class SomeBusinessClass {
  // Бизнес-данные
  // вспомогательные данные
  public void performSomeOperation(OperationInformation info) {
      // проверить уровень доступа к данным
      // проверить соответствие входных данных контракту
      // запретить доступ к данным другим потокам выполнения
      // проверить состояние кэша данных
      // занести в журнал отметку о начале операции
      // = Реализация логики данного класса =
      // занести в журнал отметку о конце операции
```

```
        // разрешить доступ к данным другим потокам выполнения
        }
}
```

В этом примере можно выделить две проблемы.

Во-первых, определяемые вспомогательные данные не относятся к требованиям, накладываемым на данный модуль, а требуются для работы сквозной функциональности.

Во-вторых, реализация performSomeOperation(..) выглядит более нагруженной, чем просто "Реализация логики данного класса".

Для правильной работы данного метода по требованиям необходимо выполнить ряд действий, не относящихся конкретно к данному модулю:

– проверить уровень доступа к данным;

– проверить соответствие входных данных контракту;

– запретить доступ к данным другим потокам выполнения;

– проверить состояние кэша данных;

– занести в журнал отметку – это требования системного уровня.

К тому же, многие из этих общих требований должны быть реализованы в других модулях.

В качестве типичного примера сквозной функциональности аспекта функционирования системы можно назвать ведение журнала событий, реализация которого присутствует во многих программных модулях. При анализе требований к системе можно выделить и другие аспекты функционирования подобного типа. Данные аспекты функционирования системы объединяет то, что их реализация не может быть локализована имеющимися средствами языка программирования в отдельном программном модуле (или ряде модулей).

Несколько признаков, которые могут указывать на проблемную реализацию сквозной функциональности при использовании существующих подходов.

Разобьем эти признаки на две категории:

– Запутанный код: в модуле может быть реализовано несколько требований. Например, часто разработчики одновременно решают проблемы связанные с бизнес логикой, производительностью, синхронизацией потоков и безопасностью. В результате множество элементов из разных требований присутствует в разрабатываемом модуле, что приводит к запутанному коду.

– Рассредоточенный код: так как сквозная функциональность по определению распространяется на множество модулей, то вызовы этой функциональности будут рассредоточены по всей системе. Например, если в системе используется требование по слежению за производительностью работы базы данных, то такая сквозная функциональность затронет все модули, работающие с базой данных.

Наличие запутанного и рассредоточенного кода влияют на проектирование и реализацию во многих отношениях:

– Плохое прослеживание назначения модуля: одновременная реализация нескольких требований в одной модульной единице делает неясным соответствие между отдельным требованием и его реализацией, в результате затруднительно понять, что реализует конкретный модуль.

– Непригодность кода для повторного использования: в связи с тем, что модуль может использовать в себе некоторую сквозную функциональность с жесткой привязкой к этой функциональности, другие части системы или другие проекты, где может потребоваться уже написанный модуль (но с другими требованиями к сквозной функциональности) не могут использовать уже написанный модуль.

– Большая вероятность ошибок: запутанность кода влечет за собой код с множеством скрытых проблем. Более того, реализация нескольких ортогональных

требований в одном модуле может привести к тому что ни одно из них не получит достаточного внимания разработчика.

– Трудность в сопровождении: появление дополнительных требований в будущем потребует переработки текущей реализации, и это может затронуть большинство существующих модулей. Модификация каждой отдельной подсистемы в отдельности под новые требования может привести к несовместимости.

Исследователи изучили различные пути выделения в отдельные модули сквозной функциональности в сложных программных системах. Аспектно-ориентированное программирование (АОП) является одним из этих решений. АОП предлагает средства выделения сквозной функциональности в отдельные программные модули – аспекты.

С точки зрения АОП в процессе разработки достаточно сложной системы программист решает две ортогональные задачи:

– разработка компонентов, то есть выявление классов и объектов, составляющих словарь предметной области;

– разработка сервисов, поддерживающих взаимодействие компонентов, то есть построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

Современные языки программирования (такие как, например, *C++*, *C#*, *Java* и т.п.) ориентированы, прежде всего, на решение первой задачи. Код компонента представляется в виде класса, т.е. он хорошо локализован и, следовательно, его легко просматривать, изучать, модифицировать, повторно использовать. С другой стороны, при программировании процессов, в которые вовлечены различные объекты, мы получаем код, в котором элементы, связанные с поддержкой такого процесса, распределены по коду всей системы. Эти элементы встречаются в коде множества классов, их совокупность в целом не локализована в обозримом сегменте кода. В результате мы сталкиваемся с проблемой "запутанного" кода.

В рамках АОП утверждается, что никакая технология проектирования не поможет решить данную проблему, если только мы будем оставаться в рамках языка, ориентированного только на разработку компонентов. Для программирования сервисов, обеспечивающих взаимодействие объектов, нужны специальные средства, возможно специальные языки. После этапа кодирования компонентов и аспектов на соответствующих языках выполняется автоматическое построение оптимизированного для выполнения (но не для просмотра и модификации) кода. Этот финальный процесс называется слиянием или интеграцией (*weaving*).

Аспектно-ориентированный подход в некоторых случаях позволяет избежать появление описанных выше проблем и улучшить общий дизайн системы, обеспечивая возможность локализации сквозной функциональности в специальных модулях – аспектах.

Основные понятия АОП [4]:

– Аспект (*aspect*) – модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом.

– Совет (*advice*) – средство оформления кода, который должен быть вызван из точки соединения. Совет может быть выполнен до, после или вместо точки соединения.

– Точка соединения (*join point*) – точка в выполняемой программе, где следует применить совет. Многие реализации АОП позволяют использовать вызовы методов и обращения к полям объекта в качестве точек соединения.

– Срез (*pointcut*) – набор точек соединения. Срез определяет, подходит ли данная точка соединения к данному совету. Самые удобные реализации АОП используют для определения срезов синтаксис основного языка (например, в *AspectJ* применяются *Java*-сигнатуры) и позволяют их повторное использование с помощью переименования и комбинирования.

– Внедрение (*introduction*) – изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код. Обычно реализуется с помощью некоторого метаобъектного протокола *MOP* (*Metaobject protocol*).

АОП позволяет реализовывать отдельные концепции в слабосвязанном виде, и, комбинируя такие реализации, формирует конечную систему. АОП позволяет построить систему, используя слабосвязанные разбитые на отдельные аспекты реализации общесистемных требований.

Разработка в рамках АОП состоит из трех отдельных шагов:

– Аспектная декомпозиция: разбиение требований для выделения общей и сквозной функциональности. На этом шаге необходимо выделить функциональность для модульного уровня из сквозной функциональности системного уровня. Например, в примере с кредитными картами можно выделить три вещи: ядро обработки кредитных карт, журнал событий, аутентификация.

– Реализация функциональности: реализовать каждое требование отдельно. В примере с кредитными картами необходимо отдельно реализовать модуль обработки кредитных карт, модуль журнала, модуль аутентификации.

– Компоновка аспектов: на этом шаге аспектный интегратор определяет правила для создания своих модулей – аспектов, составляя конечную систему. В примере с кредитными картами необходимо определить, в терминах языка реализующего АОП, при вызове каких операций необходимо вносить запись в журнал, и по завершению каких действий необходимо сообщать об успехе/неуспехе операции. Также можно определить правила, по которым будет вызываться модуль аутентификации перед доступом к бизнес-логике обработки кредитных карт.

Для иллюстрации работы интегратора аспектов вернемся к примеру обработки кредитных карт. Для краткости рассмотрим только 2 операции – кредит и дебет:

```
public classCreditCardProcessor {
  public void debit(CreditCard card, Currency amount)
    throwsInvalidCardException,
    NotEnoughAmountException,
    CardExpiredException {
        // логика по дебету
                }
  public void credit(CreditCard card, Currency amount)
    throws InvalidCardException {
        // логика по кредиту
                }
}
```

и интерфейс журнала событий:

```
public interface Logger {
  public void log(Stringmessage);
}
```

Для получения желаемой композиции требуется применение следующих правил, выраженных на обычном языке:

– записать в журнал начало каждой операции;

– записать в журнал окончание каждой операции;

– записать в журнал каждую исключительную ситуацию, которая может возникнуть в процессе работы этого модуля.

Интегратор аспектов, применяя такие правила, получит код эквивалентный данному:

```
public class CreditCardProcessorWithLogging {
  Logger _logger;
  public void debit(CreditCard card, Money amount)
      throws InvalidCardException, NotEnoughAmountException,
        CardExpiredException {
        _logger.log ("Starting CreditCardProcessor.debit(CreditCard, Money) "+ "Card: " + card + " Amount: " + amount);
        // Debiting logic
        _logger.log("Completing CreditCardProcessor.debit(CreditCard, Money) " +
```

```
        "Card: " + card + " Amount: " +
        amount);
        }
  public void credit(CreditCard card, Money
amount)
      throws InvalidCardException {
        System.out.println("Debiting");
        _logger.log("Starting CreditCardProces-
        sor.credit(CreditCard, Money) " + "Card:
        " + card + " Amount: " + amount);
        // Crediting logic
        _logger.log("Completing CreditCard-
        Processor.credit(CreditCard, Money) " +
        "Card: " + card + " Amount: " +
        amount);
        }
}
```

Автоматическая компоновка аспектов и традиционных модулей программы является ключевым свойством АОП, которое определяет основное преимущество данной технологии (рис. 2): делает возможной инкапсуляцию сквозной функциональности в отдельных программных модулях.

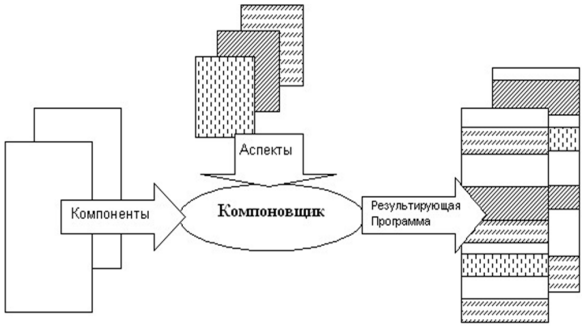

Рис. 2. Процесс компоновки аспектных и традиционных модулей

Автоматизированная компоновка аспектов и компонентов является мощным средством генерации кода и в общем случае гарантирует, что аспект будет применен ко всем модулям-компонентам, которые он затрагивает, чего сложно добиться, если вносить сквозную функциональность в модули (вручную). Реализация автоматической компоновки аспектов и компонентов во многом определяет возможности той или иной аспектно-ориентированной платформы. В настоящее время обсуждаются два подхода к интеграции аспектов: статическая интеграция на этапе компиляции и динамическая интеграция на этапе выполнения программы.

В настоящее время АОП реализуется как расширение в рамках уже существующих языков. Так, в частности, исследовательский центр *Xerox PARC* разработал систему *AspectJ*, поддерживающую АОП в рамках языка *Java*. Этот пакет встраивается в такие системы разработки, как *Eclipse*, *Sun ONE Studio*, *Forte 4J* и *Borland JBuilder*. Так как *AspectJ* является родоначальником направления АОП, используемые в нём концепции используются в качестве базовых концепций парадигмы АОП (приведены выше).

AspectJ – это простое и практическое расширение языка *Java*, которое добавляет к *Java* возможности предоставляемые АОП. Пакет *AspectJ* состоит из компилятора (*ajc*), отладчика (*ajdb*), и генератора документации (*ajdoc*).

Рассмотрим пример, на котором можно понять, как *AspectJ* реализует принципы АОП. В качестве примера возьмем модель простого графического редактора. Данный графический редактор может работать с двумя типами графических элементов – точкой и линией.

Классы, реализующие точку и линию, могут быть представлены в следующем виде:

```
class Line implements FigureElement{
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
    void moveBy(int dx, int dy) { ... } }

class Point implements FigureElement {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
    void moveBy(int dx, int dy) { ... } }
```

Классы Point и Line реализуют интерфейс FigureElement содержащий метод перемещения фигуры. Операциями, вли-

яющими на обновление экрана, являются операции перемещения фигур. После перемещения фигуры необходимо обновить изображение (*Display*). Обновление изображения в данном случае является сквозной функциональностью, которая должна вызываться при некоторых условиях (изменении положения фигур).

Язык *AspectJ* позволяет описывать несколько типов точек выполнения программы:

– вызов методов и конструкторов;

– выполнение методов и конструкторов;

– доступ к полям класса;

– обработка исключительных ситуаций;

– статическая и динамическая инициализация классов

При вызове клиентом метода moveBy все точки выполнения (*joinPoint*) находятся внутри потока управления, начиная с точки, удовлетворяющей условию. Средствами *AspectJ* можно влиять на поток управления, описав соответствующий поток управления как набор точек *pointcut*.

Далее представлено два варианта кода рассматриваемого графического редактора – с использованием *AspectJ* и без него.

Пример кода графического редактора без использования *AspectJ*:

```
class Line implements FigureElement {
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) {
        this.p1 = p1;
        Display.update(this);
  }
  void setP2(Point p2) {
        this.p2 = p2;
        Display.update(this);
  }
}

class Point implements FigureElement {
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}
```

Пример кода графического редактора с использованием *AspectJ*:

```
class Line implements FigureElement {
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}
class Point implements FigureElement {
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) {this.x = x; }
  void setY(int y) {this.y = y; }
}
aspect DisplayUpdating {
  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void FigureElement.moveBy(int, int) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)));
  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}
```

В варианте без использования *AspectJ* видно, что при изменении координат фигур в методы, отвечающие за эту функциональность, встроен код реализующий обновление дисплея. В примере с использованием аспектного подхода классы содержат код, отвечающий только за свою логику. Набор инструкций, реализующий сквозную функциональность "обновление дисплея" находится в ас-

пектном модуле – aspect DisplayUpdating. На этом примере наглядно показано основное преимущество аспектного подхода: локализация сквозной функциональности в отдельных модулях и встраивание подобной функциональности в требуемые участки системы.

Преимущества подхода с использованием АОП может показаться неочевидным на таком маленьком примере, однако следует учесть, что предложенный подход позволяет извлечь вызов аспектного метода из реализации двадцати классов так же, как и из реализации двух классов в данном примере, при этом не понадобится дописывать ни одной строки кода.

При разработке программных систем с использованием средств языка *AspectJ* можно полностью следовать трем принципам разработки аспектного подхода:

– выделять в отдельные модули сквозную функциональность (провести аспектную декомпозицию);

– реализовать каждое требование отдельно;

– интегрировать аспекты в программный код.

В примере с графическим редактором на этапе аспектной декомпозиции была выявлена сквозная функциональность – обновление дисплея. Данное требование было реализовано в аспектном модуле DisplayUpdating. В этом аспекте определяется срез точек move(..), которые включают в себя точки выполнения программы, после которых будет встроена требуемая сквозная функциональность.

Интеграции аспектов (*weaving*) происходит в момент компиляции. После компиляции получаем готовую систему с интегрированной сквозной функциональностью по правилам, описанным в аспектных модулях.

В процессе разработки программной системы средства АОП могут быть использованы в множестве таких сквозных задачах системы, как профилирование, трассировка, управление многопоточностью, кэширование, управление ресурсами, обработка ошибок и многих других.

Выводы

Рассмотрен новый подход в разработке программных систем, основное достоинство которого состоит в улучшение модульности программной системы и вытекающие из него следствия:

– выражение в явной форме структуры "сквозной функциональности";

– упрощение сопровождения и внесения изменений;

– появление новых возможностей повторного использования кода.

Необходимо отметить, что АОП не рассматривается как замена сложившимся парадигмам программирования, а исполняет роль расширения, позволяющего обеспечить модуляризацию сквозной функциональности.

В настоящее время АОП – единственная методология, позволяющая справиться со сложностью, присущей очень большим системам.

Список литературы

1. Rational Software Corporation www.rational.com

2. *Kiczales G., Lamping J., Mendhekar A., etc.* Aspect-oriented programming // ECOOP'97 : 11th European Conference on Object-Oriented Programming. – Finland, Springer-Verlag. – 9-13 June 1997. – Vol. 1241.

3. *Журавлев Е.А., Кирьянчиков В.А.* О возможности динамической интеграции аспектов в аспектно-ориентированном программировании // Изв. СПбГЭТУ (ЛЭТИ) Серия: Информатика, управление и компьютерные технологии. – 2002. – Вып. 3. – С.81–86.

4. Википедия. Свободная энциклопедия: http://ru.wikipedia.org/wiki/Аспектно-ориентированное_программирование.